CS-E4890: Deep Learning
Q&A Session Assignment 5 - Transformers

Lukas Prediger

31.03.2023

Your tasks

You will have to implement the same translation task as in the previous assignment, but this time using a transformer architecture. This is broken down into the implementation steps

1. the `collate` function for converting input data batches to PyTorch tensors,
2. `EncoderBlock` and `Encoder` modules,
3. `DecoderBlock` and `Decoder` modules,
4. the training loop,
5. the `translate` function to perform translation of any source sentence without knowing the target.

Topics of this session

1. Recap of the transformer architecture
2. Batching, padding and `collate`
3. Common implementation pitfalls & tips
4. Your questions

Transformer Recap

The goal

je suis impatiente de te voir danser .

ENCODER

EMBEDDED MEANING

DECODER

i m looking forward to seeing you dance .

Decoder is autoregressive with contextual information from encoder

ENCODER

i m looking forward to seeing you ...

z

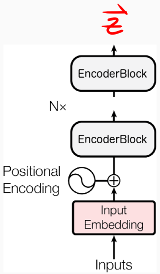
DECODER

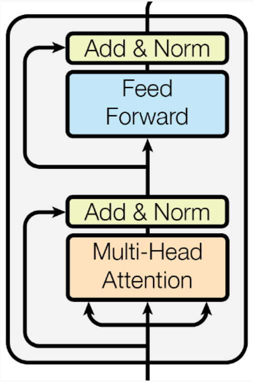
i m looking forward to seeing you dance .

Decoder is autoregressive with contextual information from encoder

SOS i m looking forward to seeing you dance

i m looking forward to seeing you dance .


EncoderBlock
Nx
EncoderBlock
Positional
Encoding
Input
Embedding
Inputs


Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention

Embedding

je suis impatiente de te voir danser .

WORD EMBEDDING

POSITIONAL ENCODING

i think that i could ...

Attention

je suis impatiente de te voir danser .

1 2 3 4 5 6 7 8

$\vec{s}_1$ $\vec{s}_2$ $\vec{s}_3$

$$\vec{s}_1{}' = \sum_{j=1}^{8} w_{1j}\, \vec{s}_j$$

VALUES

$$\sum_{j=1}^{8} w_{1j} \overset{!}{=} 1$$

$$w_{1j} \approx \text{SIMILARITY}(\vec{s}_1, \vec{s}_j) = \vec{s}_1^{\,T} \vec{s}_j =: a_{1j}$$

QUERY KEY

$$w_{1j} = \frac{e^{a_{1j}}}{\sum_i e^{a_{1i}}}$$

Attention

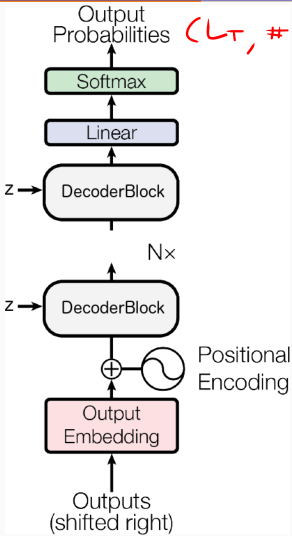
KEYS

QUERIES

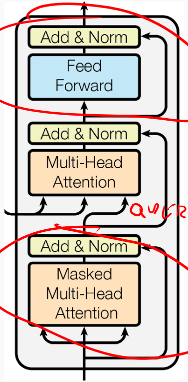
		1	2	3	4	5	6	7	8	9	10
		$\vec{s}_1$	$\vec{s}_2$	$\vec{s}_3$	$\vec{s}_4$	$\vec{s}_5$	$\vec{s}_6$	$\vec{s}_7$	$\vec{s}_8$	$\vec{s}_9$	$\vec{s}_{10}$
1	$\vec{s}_1$	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	$w_{1,6}$	$w_{1,7}$	$w_{1,8}$	$w_{1,9}$	$w_{1,10}$
2	$\vec{s}_2$	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	—	—	—				
3	$\vec{s}_3$	$w_{3,1}$	$w_{3,2}$	-	—	—					
⋮					⋮						

$\cdot \; V \rightarrow \vec{s}_1\,' \; \dots \; \vec{s}_8\,'$

Overall architecture: Decoder



(LT, #words in target language)



queries

Decoder Self-Attention

SOS i m looking forward to seeing you dance

i m looking forward to seeing you dance .

$M \in \mathbb{R}^{L_T \times L_T}$

$A \oplus M$

KEYS

QUERIES

$$
\begin{array}{c|ccccccccccc}
 & \vec{t}_1 & \vec{t}_2 & \vec{t}_3 & \vec{t}_4 & \vec{t}_5 & \vec{t}_6 & \vec{t}_7 & \vec{t}_8 & \vec{t}_9 & \vec{t}_{10} & \vec{t}_{11} \\
\hline
\vec{t}_1 & A_{1,1} & \cancel{A_{1,2}} & \cancel{A_{1,3}} & \cancel{A_{1,4}} & \cancel{A_{1,5}} & \cancel{A_{1,6}} & \cancel{A_{1,7}} & \cancel{A_{1,8}} & \cancel{A_{1,9}} & \cancel{A_{1,10}} & \cancel{A_{1,11}} \\
\vec{t}_2 & A_{2,1} & A_{2,2} & \cancel{A_{2,3}} & \cancel{A_{2,4}} & \cancel{A_{2,5}} & \cancel{A_{2,6}} & \cancel{A_{2,7}} & \cancel{A_{2,8}} & \cancel{A_{2,9}} & \cancel{A_{2,10}} & \cancel{A_{2,11}} \\
\vec{t}_3 & A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & A_{3,5} & A_{3,6} & A_{3,7} & A_{3,8} & A_{3,9} & A_{3,10} & A_{3,11} \\
\vdots & & & & & & & \vdots & & & &
\end{array}
$$

$-\infty$

0

0

$A \in \mathbb{R}^{L_T \times L_T}$

Decoder Cross-Attention

$\vec{s}_1 \; \ldots \; \vec{s}_8$ ← KEYS, VALUES

$\vec{t}_1 \; \ldots \; \vec{t}_{4T}$ ← QUERYS

Batching, padding, `collate`

Batching

Transformers are (also) about efficiency: We want to parallelise as much as possible.

Since sequences have different lengths, we need to pad them in order to create a tensor!

Batching

je suis impatiente de te voir danser . PAD

il est toujours en train de se plaindre .

Padding and Attention

(B, L)

je suis impatiente de te voir danser . PAD

1 2 3 4 5 6 7 8 9

p_1 — — — — — — p_9

$w_{19} \stackrel{!}{=} 0$ $w_{i9} \stackrel{!}{=} 0$

$$p_j = \begin{cases} 0 & \text{if padding} \\ 1 & \text{otherwise} \end{cases}$$

$$w_{ij} = \frac{e^{a_{ij}} \cdot p_j}{\sum_{k=1} (e^{a_{ik}} \cdot p_k)}$$

Common implementation pitfalls

Incorrect use of `tr.PositionalEncoding`.

We provide `tr.PositionalEncoding` to you for the positional encoding step.

`tr.PositionalEncoding` adds the positional encoding to the input it is provided *internally*

you do not need to perform the addition step yourself!

Including padded positions in the loss...

... incentivises the model to become really good at predicting padding.

At the expense of becoming less good at predicting interesting things.

Solution: PyTorch loss functions (`nll_loss`, `cross_entropy`, ...) have a `ignore_index` parameter.

`nn.Embedding` similarly has a `padding_idx` parameter.

Not shifting the target sequence...

... causes the decoder to learn to predict the *current* instead of the *next* word.

Some other tips

- `nn.Sequential` simplifies MLP implementation: https://pytorch.org/docs/1.10/generated/torch.nn.Sequential.html#torch.nn.Sequential.
- `nn.ModuleList` helps with blocks in `Encoder` and `Decoder`: https://pytorch.org/docs/1.10/generated/torch.nn.Sequential.html#torch.nn.ModuleList.
- Consider creating additional `nn.Module` classes for reusable sub-blocks.

Questions?

Room for questions